(19) Octrooiraad Nederland

(11) Publikatienummer: **9100598**

(12) **A TERINZAGELEGGING**

(21) Aanvraagnummer: **9100598**

(22) Indieningsdatum: **05.04.91**

(51) Int.Cl.[5]: **G06F 9/38, G06F 15/82**

(43) Ter inzage gelegd: **02.11.92 I.E. 92/21**

(71) Aanvrager(s): **Henk Corporaal, Vogeldreef 22 te 2727 AM Zoetermeer en Johannes Martinus Mulder, Karel Doormanstraat 268 te 3012 GN Rotterdam; wier gemeenschappelijk domicilie is: Henk Corporaal, Vogeldreef 22 te 2727 AM Zoetermeer**

(72) Uitvinder(s):

De uitvinder heeft van tenaamstelling afgezien.

(74) Gemachtigde: **Geen**

(54) **MOVE, een flexibele en uitbreidbare architectuur voor hoge prestatie processor ontwerp**

(57) Afgelopen decennium zagen we een algemene acceptatie van de RISC-ontwerp-principes. Om nog betere prestaties te verkrijgen, is veel onderzoek gericht op het ontwikkelen van architecturen die instructieniveau-parallellisme exploiteren. Twee eigenschappen onderscheiden deze architecturen: 1. hoe wordt parallellisme gedetecteerd (statisch of dynamisch), en 2. hoe wordt het geïmplementeerd (pipelined versus meerdere functie-eenheden). Deze architecturen hebben echter een aantal nadelen welke resulteren in een complexe organisatie en inefficiënt gebruik van hardware. VLSI wordt niet tot de uiterste grenzen geëxploiteerd wat betreft optimale cyclustijd, en verder hebben deze architecturen onvoldoende flexibiliteit voor het op eenvoudige wijze genereren van applicatie-specifieke-processoren.

Ter vermijding van bovengenoemde nadelen hebben wij een architectuur uitgevonden die eenvoudig te maken is, de beschikbare hardware efficiënter gebruikt, optimale cyclustijd toelaat, en die eenvoudig te reconfigureren is voor andere functionaliteit en prestatie. Naast bruikbaarheid voor algemene berekeningen, is deze architectuur bijzonder geschikt als raamwerk voor het semi-automatisch genereren van processoren voor applicatie-specifieke-doeleinden waar zeer hoge prestaties verlangd worden.

De essentie van deze uitvinding is een architectuur welke het datatransport volledig scheidt van de operaties op deze data, waardoor de aanwezige transportcapaciteit veel beter gebruikt wordt. Deze scheiding geeft onder andere zeer veel vrijheid aan de implementatie van functie-eenheden; zo kunnen verschillende pipelining-schema's door elkaar gebruikt worden. Verder kunnen we de cyclustijd reduceren tot het absolute minimum, namelijk de tijd benodigd voor het transport van data. Het resulterende programmeermodel is ongebruikelijk en geeft scheduling-mogelijkheden welke in voorgaande architecturen niet voorkomen.

NL A 9100598

MOVE: Een Flexibele en Uitbreidbare Architectuur voor het Ontwerpen van Processoren

1 Inleiding

Een van de meest belangrijke ontwikkelingen in het ontwerp van computer systemen was het gebruik van RISC- i.p.v. CISC-ontwerpprincipes [1]. De belangrijkste les was, dat extra hardware (bijv. voor het implementeren van complexe instructies) niet altijd resulteert in snellere executie; in tegendeel, het kan de totale executie tijd doen toenemen. Mogelijke oorzaken hiervoor zijn:

1. Extra hardware functionaliteit kan het kritieke tijdspad vergroten, hetgeen tot een verhoogde cyclustijd leidt.
2. Complexe instructies worden in algemene berekeningen zelden gebruikt. Hardware hieraan besteedt kan wellicht effectiever voor andere verbeteringen aangewend worden.
3. Complexe instructies zijn moeilijk te *pipelinen*.
4. Soms zijn software oplossingen voor complexere instructies zelfs sneller dan de overeenkomstige hardware oplossing; i.h.b. indien compiler optimalizaties de software overhead voor een deel elimineren.
5. Complexe hardware verhoogt de ontwikkeltijd en maakt daardoor het gebruik van de laatste technologische verbeteringen onmogelijk.

Afgezien van deze oorzaken, vermindert de implementatie van complexe functies de ontwerp-flexibiliteit. Deze functies kunnen het ontwerp zodanig bepalen, dat het aanbrengen van toekomstige verbeteringen lastig wordt.

Momenteel leveren de meeste CPU-fabrikanten op RISC-principes gebaseerde computersystemen met vergelijkbare prestaties. De prestatie wordt gedomineerd door de gebruikte technologie. Daar de basis complexiteit van een RISC-processor tamelijk beperkt is, kan deze eenvoudig worden geïntegreerd in een VLSI-ontwerpomgeving voor de creatie van applicatie-specifieke-processoren ([2]). Theoretisch haalt een RISC-systeem een prestatie van 1 CPI (cyclus per instructie); in de praktijk is deze iets groter t.g.v. penalties veroorzaakt door *load* en *branch* opdrachten. Efficiënt gebruik van RISC-systemen vereist compilers welke in staat zijn deze penalties te minimalizeren d.m.v. het plaatsen van instructies in *delay slots*; dit kan beschouwd worden als een beperkte vorm van *scheduling* van parallelle instructies.

Ter verkrijging van een nog grotere performance verbetering, dan alleen met snellere technologie mogelijk is, onderscheiden we twee belangrijke architectuur-technieken: 1) het gebruik van diepe *pipelines*, en 2) het gebruik van meerdere onafhankelijke functie-eenheden (FUs). De eerste techniek reduceert de cyclustijd, de tweede de CPI. Beide technieken vereisen uitgebreide parallellizatie van code op het instructie-niveau. De volgende sectie bespreekt deze ontwikkelingen. Tevens wordt verklaart waarom de resulterende architecturen een aantal nadelen hebben, welke hun toepasbaarheid sterk beperken.

De hier beschreven uitvinding betreft een zeer afwijkende architectuur, genoemd de MOVE-architectuur, welke het data-transport en de data-operaties volledig scheidt. Sectie 3 beschrijft deze uitvinding terwijl sectie 2 de recente architectuur-ontwikkelingen en de MOVE-concurenten beschrijft. Sectie 4 vergelijkt de MOVE-architectuur met deze concurrerende architectuur-implementaties en toont de superioriteit van de beschreven uitvinding. Sectie 5 bevat de conclusies, sectie 6 de figuren en de laatste sectie bevat een verklarende woordenlijst en een bibliography.

2 Recente Architectuur-ontwikkelingen

Om de snelheid van enkelvoudige processoren te verhogen worden verschillende architectuur-technieken toegepast [3], te weten:

Superpipelining: Het opdelen van bestaande *pipeline stages* in *sub-stages*, waardoor een kortere cyclustijd mogelijk wordt. Dit is een uitbreiding van het RISC-*pipeline*-principe; daar besloeg de executie-*stage* nog 1 cyclus, terwijl deze nu wordt opgedeeld in meerdere stages. *Superpipelining* zal de effectieve CPI iets vergroten tengevolge van het toegenomen aantal *delay slots*, echter dit effect wordt gecompenseerd door de verlaagde cyclustijd. Voorbeelden zijn te vinden in [4,5].

Functioneel parallellisme: Door het toevoegen van onafhankelijke functie-eenheden ter exploitatie van het operatie parallellisme heeft als gevolg dat de effectieve CPI lager wordt dan één. *Superscalars* [6,7] en *VLIWs* [8,9,10,11,12] zijn vertegenwoordigers van het toepassen van deze techniek. *Superscalars* passen dynamische detectie van operatie-parallellisme toe, terwijl VLIWs dit volledig door de compiler (statisch) laten doen. Hoewel *Superscalars* ter verkrijgen van een redelijke efficiënte executie ook intensieve compiler ondersteuning vereisen, kunnen zij ook *niet-geschedulede* code uitvoeren; zij kunnen daardoor *object-code* compatibel zijn met architecturen die niet met functioneel parallellisme zijn uitgevoerd.

Superscalars hebben een sterke beperking in de hoeveelheid te exploiteren parallellisme. De hardware nodig voor *runtime* parallellisme-detectie beperkt de grootte van het *decode-window* tot een paar instructies. Verdere hardware reductie is mogelijk door het aantal mogelijk te combineren instructies te beperken (bijv. maximaal 1 integer, 1 drijvende-komma, 1 *load/store* en 1 controle-instructie). De compiler is veel beter in staat om mogelijke parallelle-instructies te ontdekken dan met hardware ooit mogelijk is (een mogelijke uitzondering is het detecteren van adres-afhankelijkheden bij geheugen-operaties). Een gunstig gevolg van dynamische detectie is wel, dat de grootte van de code niet toeneemt t.g.v. het invoegen van lege instructies (NOPs). Kortom, superscalars zijn goed voor beperkte prestatie verbeteringen van bestaande systemen, maar vereisen hiervoor wel compiler support.

VLIW- en *Superpipelined*-architecturen hebben een hoger potentieel m.b.t. parallelle executie dan Superscalars. Het voordeel zal voor scalaire applicaties beperkt zijn (i.v.m. de sequentiële aard van de berekeningen); echter voor specifieke- en vector-applicaties kan een veel grotere snelheidstoename gerealiseerd worden.

Idealiter willen we *VLIW*- en *Superpipelining*-technieken combineren. Dit zou een architectuur opleveren die hoge vector-prestaties levert, die geschikt gemaakt kan worden voor specifieke-applicaties en ook nog een goede scalaire-prestatie laat zien. Echter, zoals we zullen aantonen in sectie 4, bezit deze architectuur een aantal nadelen welke resulteren in een complexe-organizatie, inefficient hardware gebruik, moeilijk te wijzigen functionaliteit, en beperkte uitbreidbaarheid. Daardoor is zij niet ideaal voor gebruik in een raamwerk voor applicatie specifiek processor ontwerp. De uitvinding welke beschreven wordt in de volgende sectie heeft deze nadelen niet.

3 De MOVE-architectuur

De vereiste prestatie, en de toegestane ontwerp en fabricage kosten bepalen voor VLIW het aantal en type en voor Superpipelined de diepte en type van de te gebruiken functie-eenheden. De communicatie-bandbreedte tussen functie-eenheden wordt uitsluitend bepaald door het aantal en type functie-eenheden; zij dient op de *worst-case* situatie berekend te zijn. Evenzo is de instructie-bandbreedte afgestemd op de *worst-case* situatie, n.l. dat alle functie units tegelijkertijd in gebruik zijn. Zelfs indien dit het geval is, zal de vereiste communicatie-bandbreedte zelden *worst-case* zijn.

In deze sectie beschrijven we een uitvinding van een architectuur (de MOVE-architectuur), die bovengenoemde problemen van *overdesign* sterk reduceert. In de volgende subsectie ontwikkelen we deze architectuur vanuit het gezichtspunt van VLIW en Superpipelining architecturen. Vervolgens worden zaken als *pipelining*, het afhandelen van excepties, en de benodigde ondersteunings-mechanismen, nodig voor een efficiënte afbeelding van hogere programmeertalen en operating-systemen, besproken.

3.1 Van VLIW en Superpipelining naar MOVE

Indien we VLIW en Superpipelining als vertrekpunt kiezen, kan de ontwikkeling van de MOVE-architectuur als een 6-staps proces worden gezien: 1) reductie van register-file communicatie-bandbreedte, 2) het programmeren van de *bypass*, 3) het reduceren van de *bypass*-connectiviteit, 4) het separeren van transport en operatie, 5) reductie van de tranport capaciteit en 6) het programmeren van het transport.

Reductie van de register-file-communicatie-bandbreedte Figuur 1a toont de interne communicatie-structuur van een (*super-*) *pipelined* organizatie. Een set van functie-eenheden (FUs) is verbonden met een bypass-eenheid welke operand- en bypass-registers bevat. De bypass-eenheid is verbonden met een register-file. Op soortgelijke wijze toont figuur 1b de communicatie-structuur van een VLIW. Deze VLIW organizatie wordt in het vervolg als uitgangspunt gebruikt.

De bypass bevat naast operand-registers, een aantal bypass-registers voor tijdelijke resultaten. Deze laatste zijn georganizeerd als *FIFOs*. De operand-registers lezen hun data uit de register-file, de FIFO, of de uitgangen van de FUs. Het schrijven van resultaten kan in alle FIFO registers geschieden, afhankelijk van de tijd die een FU-operatie duurt. Dit schrijven gaat vergezeld van het schrijven van de register-*identifier*. Deze identifier dient voor het schrijven in de register-file en voor identificatie van de tijdelijke resultaten in de bypass. Een resultaat welke de FIFO verlaat wordt in de register-file geschreven. De bypass is niet zichtbaar op architectuur niveau.

Met respect tot de benodigde bandbreedte tussen de register-file en de bypass kunnen we een 2-tal observaties geven:

1. Meer dan 50 % van de data in de bypass-registers wordt na het schrijven hiervan in de register-file niet meer gebruikt. Diepere FIFOs kunnen dit percentages nog verhogen.
2. Veel instructies gebruiken niet alle mogelijke register-operanden. Bijv. een *branch* instructie genereert geen resultaat; monadische instructies gebruiken maar 1 source-operand. Deze instructies gebruiken dus niet de volledige register-file bandbreedte.

De compiler kan al deze gevallen van onvolledig gebruik van register-bandbreedte detecteren. We kunnen daardoor een organizatie construeren waarbij het register-transport volledig door de compiler gecontroleerd wordt. Zoals figuur 2a laat zien, wordt de register-file nu een functie-eenheid met veel minder lees- en schrijf-poorten.

Programmeren van de bypass In de nu gecreëerde situatie is het efficiënter om de bypass zichtbaar te maken op architectuur niveau. De bypass wordt nu een expliciet geadresseerd hoogste niveau van de geheugen hiërarchie. Het schrijven van waarden in de register-file wordt nu niet meer beïnvloed door de FIFOs; het is volledig onder controle van de compiler. Alle bypass-registers (vanaf nu inclusief de operand-registers) kunnen beschreven en gelezen worden door alle FUs (zie figuur 2b). De voordelen hiervan zijn:

- Eenvoudiger ontwerp van de register-file.
- Eenvoudiger ontwerp van de bypass (geen FIFOs, geen *identifier matching*).

- Kleinere operand-identifier velden in de instructies.

Reductie van de bypass-connectiviteit Daar lezen en schrijven van de bypass-registers onderdeel is van de FU-executie, dient dit zo efficiënt mogelijk te geschieden. Een manier om dit te verwezenlijken is het reduceren van het aantal connecties en daarmee van de *read-* en *write-load.* Toepassing van dit principe op de zichtbare bypass voor het aantal lees- en schrijf-connecties resulteerd in de organizaties getekend in de figuren 2c en 2d resp.

De eenvoudigste methode ter vermindering van het aantal lees-connecties is het gebruik van privé register-sets voor alle FU-ingangen. Deze sets kunnen door alle FUs beschreven worden (zie figuur 2c). De LIFE-processor [13] gebruikt deze organizatie. Ter vermijding van meerdere schrijfpoorten per register-set beperkt LIFE het schrijven per register-set tot één FU per keer.

Een andere methode is het reduceren van het aantal schrijf-connecties. Iedere FU schrijft hierbij in een eigen bypass-register. De Cydra-5 [12] gebruikt deze organizatie. Figuur 2d toont een organizatie met 2 privé resultaat registers per FU.

Separatie van transport en operatie Reductie van zowel het aantal lees- en schrijf-connecties wordt geïllustreerd in figuur 3a. Alle bypass-registers zijn nu ingedeeld als privé FU-resultaat en FU-operand registers. De gehele operatie vindt nu in de FU plaats, het transport daarbuiten. Transport en operatie zijn daardoor gescheiden.

Reductie van transport-capaciteit Het transport-netwerk uit figuur 3a wordt nog steeds niet ten volle benut; immers, niet alle FUs zullen iedere cyclus een resultaat afleveren. De transport-capaciteit van dit netwerk moet beter worden afgestemd op gemiddeld gebruik; d.w.z. het aantal bussen moet worden gereduceerd (of gelijkwaardig, met hetzelfde aantal bussen kunnen meer FUs bediend worden). Figuur 3b toont 3 FUs die samen 2 bussen moeten delen.

De belangrijkste implicatie van deze organizatie is dat we zowel operaties als transport moeten *schedulen.* Gezien vanuit VLIW gezichtspunt worden de operaties door de compiler (statisch) en het transport-netwerk door de hardware (dynamisch) *gescheduled.*

Programmeren van het transport De volgende logische stap is om ook de compiler het tranport te laten *schedulen.* De compiler kan hier potentieel een veel betere prestatie leveren dan de hardware; i.t.t. hardware, kan de compiler het gehele programma bekijken. Nu wordt het transport zichtbaar op architectuur niveau. Dit betekent dat operaties niet apart gespecificeerd behoeven te worden, zij treden op als zij-effect van het transport. Als gevolg is het programmeer-model omgedraaid:

Traditioneel:	Geprogrammeerde operations, operaties triggeren transport.
MOVE:	Geprogrammeerd transport, transport triggert operaties.

De MOVE-architectuur is gebaseerd op dit concept; zij kan worden gezien als een niet-uniforme register-set verbonden d.m.v. een *point-to-point* of *point-to-multipoint* of ander type netwerk [1]. De register-set bevat FU-input en FU-output registers, en *general-purpose* registers. De input-registers zijn verdeeld in 2 types: operand- en trigger-registers. Schrijven van data in het trigger-register start een FU-operatie, waarbij de

[1]Het gebruik van uitsluitend move-instructies is niet nieuw (zie o.a. [14]). Move-architecturen, waarbij alle functionaliteit *memory-mapped* is, zijn reeds lang bekend. Echter het gebruik van een volledige *register-mapped* functionaliteit, waarbij meerdere guarded tranport-opdrachten parallel kunnen worden uitgevoerd is nieuw. Deze methode leidt tot allerlei verrassende voordelen zoals in sectie 4 beschreven.

data uit een aantal (0 of meer) FU-specifieke operand-registers de overige input-operanden bevatten. Als gevolg bevat de MOVE-architectuur maar 1 type instructie: de *move*. Een *move* bevat tenminste een *source-* en een *destination-identifier*. Het specificeert een transport van een register (of FU) naar een ander (of meerdere i.g.v. *multicast* transport) register (of FU), waarbij mogelijkerwijs 1 of meerdere operaties worden getriggerd.

Voor een traditionele optel-operatie heeft de MOVE-architectuur 3 moves nodig, de eerste om het operand-register te laden, de tweede voor het triggeren van de operatie (waarbij de tweede source-operand getransporteerd wordt) en de derde voor het lezen van het resultaat. In de praktijk zal deze laatste move veelal ook de volgende operatie triggeren.

Het transport-netwerk kan een willekeurige topologie hebben, zolang transport-opdrachten kunnen worden uitgevoerd. Het netwerk behoeft geen volledige connectiviteit. Het is de taak van de compiler het transport te optimalizeren voor een gegeven topologie. Beperkte connectiviteit kan de capaciteit en daardoor de cyclustijd verlagen. Het MOVE-prototype, welke de MOVE-architectuur realiseert op een *single chip*, bevat een reguliere volledig verbonden bus-structuur, waarbij 4 move-opdrachten parallel worden uitgevoerd.

3.2 Pipeline alternatieven

De MOVE-architectuur vereist niet dat de FUs gepipelined moeten worden uitgevoerd, echter optimaal FU-gebruik vereist maximale FU-pipelining. Indien we *pipeline-latches* met logica combineren kunnen we het aantal poorten per *pipeline-stage* reduceren tot 2. In de praktijk (zie [15]) kunnen *clock-skew* en *data-skew* een hogere ondergrens opleveren. Bij het implementeren van pipelines hebben we een aantal alternatieven:

- **Continue always** In dit geval wordt de data in de pipeline op reguliere intervallen (meestal iedere klokcyclus) naar de volgende *stage* verschoven. Dit is geïmplementeerd in de Cydra-5 VLIW processor [12] en in de meeste vector-machines. Dit is eenvoudig te implementeren, echter het vereist, zoals verderop zal worden aangetoond, veel extra registers.
- **Push/Pull** Hier wordt de data in de pipeline alleen doorgeschoven in geval een volgende instructie dezelfde pipeline gebruikt [5]. Merk op dat het onderscheid tussen *push* of *pull* alleen betekenis heeft indien traditionele operaties worden opgesplitst in afzonderlijke *moves*.
- **Hybrid** Deze oplossing houdt in dat data in de pipeline meestal mag doorschuiven, behalve in het geval data afkomstig van eerdere operaties dreigt overschreven te worden, doordat bijv. de resultaten niet optijd zijn gelezen. Pipeline-stages blokkeren dus zo laat mogelijk.

In de MOVE-architectuur hebben de meeste FUs een hybride pipeline-implementatie. De eerste oplossing vereist dat de resultaten exact op de juiste tijd worden gelezen. Dit vermindert niet alleen de *scheduling*-mogelijkheden van de MOVE-machine, doch betekent verlies van data in geval van blokkeren (denk aan *cache-miss* of een exceptie). Dit is niet het geval als de pipeline slechts *1-stage* diep is.

Het tweede alternatief leidt tot grote code omvang voor scalaire applicaties; extra dummy instructies zijn nodig, alleen om resultaten uit de pipeline te halen.

Het hybride alternatief is iets moeilijker te implementeren omdat *stages* conditioneel mogen doorschuiven, echter het combineert eenvoudige afhandeling van excepties, kleine code-omvang en extra scheduling-vrijheden (zie sectie 4).

3.3 Excepties

Excepties kunnen worden onderverdeeld in drie categoriën:

1. **Precies.** Het behandelen van deze excepties vereist dat de source-operanden van de operatie die aanleiding geeft tot de exceptie nog voorhanden zijn. Voorbeelden zijn *loads*, *stores* en drijvende-komma-operaties (volgens de IEEE 754 floating-point standaard). Na behandeling van de exceptie moet de executie voortgezet kunnen worden.
2. **Niet-precies.** Deze excepties vereisen niet dat de source-operanden nog aanwezig zijn voor de behandeling van de exceptie.Ze vereisen echter wel dat de executie na behandeling voortgezet kan worden. Voorbeelden zijn excepties veroorzaakt door integer-arithmetiek[2].
3. **Fataal.** In contrast met de vorige twee wordt voor deze exceptie niet de eis gesteld dat executie weer voortgezet moet kunnen worden.

Preciese excepties worden gedeeltelijk geïmplementeerd doormiddel van het gebruik van de zo genoemde inexacte exceptie-conditie. FUs proberen zo snel als mogelijk te verifieren of mogelijk een exceptie kan optreden. Voordat deze conditie geverifiëerd is en ook indien deze conditie waar is, moeten we er voor zorgen dat de source-operanden niet worden overschreven. Dit is op vier manieren te realiseren: 1) door middel van compiler-afspraken, 2) door onmiddelijk *locking* te initiëren totdat bekend is dat de inexacte conditie niet geldig is, of totdat het bekend is dat de exceptie echt is opgetreden, 3) door locking te initiëren wanneer de source-operanden dreigen te worden overschreven, of 4) door het redden van de source-operanden in de FU zelf. Het zal duidelijk dat oplossing 3, samen met een snelle generatie van de inexacte conditie te prefereren is. Voor de meeste FUs uit het MOVE-prototype is dan ook deze combinatie geïmplementeerd.

Om van een exceptie te kunnen herstellen dient de processor in een staat te worden gebracht die een mogelijk herstel garandeert. Hier zijn grofweg drie methoden voor: 1) *restart*, 2) *completion*, 3) *blocking*.

Restart In het geval van multi-cycli operaties met *out-of-order* beëindiging is het herstarten (*restart*) van onafgemaakte instructies praktisch onmogelijk. De PC en overige *processor-state* dient in dat geval voor vele cycli bewaard te blijven, hetgeen een enorme hardware investering vergt.

Completion Veel *superpipelined* en VLIW processoren zijn ontworpen om na een exceptie die werkzaamheden af te maken waar de verschillende FUs mee bezig zijn (*completion*). Deze processoren hebben niet de mogelijkheid de FUs te blokkeren tijdens excepties[3]. Het gevolg hiervan is, dat al de onafgemaakte instructies na het detecteren van de exceptie alsnog worden afgemaakt. Dit impliceert op zich dat de compiler is gedwongen om de resultaat-registers van multi-cycli operaties niet te gebruiken voor andere operaties tijdens de berekening van deze multi-cycli operatie. In het geval dat excepties precies gedetekteerd moeten worden, wordt dit inefficient register-gebruik alleen maar erger: ook de source-operanden moeten nu beschermd worden tegen overschrijving tijdens de operatie op deze operanden. In de Cydra-5 [12] garandeert de compiler dit, door een register als *in gebruik* te verklaren vanaf het begin van de operatie tot het moment waarop al de operaties die de geschreven waarde gebruiken afgelopen zijn. Voor *loops* die *software-pipelined* zijn geeft dit een vrij disastreuse toename in registergebruik. Vergeet niet dat al de register-*files* in Cydra-5 ontworpen moeten worden voor *worstcase* situaties.

Blocking In de MOVE-architectuur blokkeren de FUs vanzelf i.g.v. een exceptie, omdat dit mechanism reeds is ingebouwd om een extra graad van *scheduling*-flexibiliteit te verkrijgen. De MOVE-*pipelines* blokkeren vanzelf wanneer het resultaat niet gelezen wordt. Register bestemmingen worden daarom nooit overschreven. In feite kan een bestemming niet overschreven worden, omdat de FUs niet weten wat

[2]Sommige LISP systemen kunnen preciese excepties voor integer-arithmetiek goed gebruiken. Als gevolg van een *overflow* op *fixnum* operaties kan dan de representatie naar *bignum* gewijzigd worden.

[3]Een uitzondering hierop is Intel i860 [5]. De *pipelines* van de i860 moeten doormiddel van duw-en-trek instructies geleegd worden.

(buiten het FU-resultaat register) de uiteindelijke bestemming is. Het redden en herstellen van de FU-*state* kan bijzonder complex zijn in de huidige VLIWs. Drie redenen hiervoor zijn: 1) *stages* kunnen complexe data formaten hebben, 2) bestemming identificatie dient bewaard en hersteld te kunnen worden en 3) de data en bestemming-identificatie dient te worden hersteld in de juiste *pipeline-stage*. Voor de MOVE-architectuur is *blocking* veel makkelijker te implementeren. 1) het is niet nodig tussenresultaten te bewaren, slechts eindresultaten zijn nodig. 2) omdat de instructies gesplitst zijn in componenten voor het laden van operanden en het lezen van resultaat, is de enige bestemming het FU-resultaat register en is het dus niet nodig de bestemming-identificatie te bewaren. 3) aangezien het aantal cycli tussen het starten van een operatie en het lezen van de resultaten niet bepaald wordt door de FU-*latency* is slechts het herstellen van de juiste volgorde van operaties in een FU belangrijk; de exacte positie in de FU-*pipeline* is irrelevant. De FU behoeft dus ook geen complexe organisatie om dit herstel proces mogelijk te maken, slechts een eenheids-operatie per FU is voldoende om herstel te bewerkstelligen (bijv. +0, x1, enz.).

3.4 Support-mechanismen

De MOVE-architectuur separeert transport van operatie en wordt geprogrammeerd door uitsluitend het transport te specificeren. Een processor gebasseerd op deze architectuur vereist een aantal support-mechanismen ter ondersteuning van hogere programmeertalen en operating-systemen. Figuur 4 toont een mogelijke realizatie van zowel het move- als de support-mechanismen, zoals dit in het prototype gerealiseerd is. Het move-mechanisme bestaat uit het transport-netwerk en de move-identifier-bus. De support-mechanismen zijn de guard-, locking- en exceptie-mechanismen.

Conditionele executie Het uitsluitend ondersteunen van conditionele executie d.m.v. een *compare-* en *branch-mechanisme* is minder acceptabel voor VLIW en superpipelined machines. Een betere aanpak is het gebruik van *guards*, welke conditionele executie van een aantal of alle transport-operaties mogelijk maken. Een aantal (of alle) moves bevatten een *guard-selector*, welke de waarde van een guard als conditie selecteert, of een combinatie (boolean-expressie) van deze guards specificeert. De semantiek van een *move* is zodanig, dat indien de guard *false* is, de move-operatie geen gevolgen heeft [4].

Het guard-mechanisme is geïntegreerd in een FU. Deze FU verwerkt de guard-selector zoals gespecificeerd in de move-instructies. Voor iedere gespecificeerde guard-selector activeert deze FU de corresponderende *guard-lijn*. Deze FU implementeert tevens de operaties voor het zetten van de guards. Het guard-mechanisme in combinatie met de scheiding van *trigger* en *result* move maakt het speculatief opstarten van een FU-operatie eenvoudig. Het verwijderen van ongewenste resultaten (eventueel inclusief ongewenste exceptie conditie) gebeurt in het prototype met behulp van een move naar een *dummy* register.

Locking Compile-time synchronizatie van alle move-operaties is niet mogelijk indien *FU-latencies* niet door de compiler kunnen worden bepaald (e.g. in geval van een cache-miss), en is niet efficient indien onvoldoende operaties kunnen worden gevonden om de *FU-latency* te overbruggen i.g.v. een *RaW-hazard*. Beide problemen worden opgelost door hardware synchronizatie. In het algemeen impliceert synchronizatie dat producent en consument elkaar kunnen *locken*, indien één van tweeën nog niet klaar is voor de transactie. *Locking* kan ook impliciet plaatsvinden indien het transportnetwerk uitgevoerd wordt met behulp van *selftimed* logica.

In de MOVE-architectuur leidt *locking* tot 2 types transport-locks: 1) een lees-lock, zolang een resultaat (in een FU) nog onderweg is, en 2) een schrijf-lock, zolang de FU nog niet beschikbaar is. De lees-lock maakt het *schedulen* van resultaat-moves onafhankelijk van de FU-latency, de schrijf-lock staat een

[4] Een mogelijke uitzondering, die wij niet in het prototype geïmplementeerd hebben, is dat deze move toch een waarde uit een FU-pipeline leest, doch hiermee niets doet, en ook mogelijke excepties onderdrukt; dit ter ondersteuning van speculatieve executie.

efficiënte implementatie van preciese excepties toe. *Locking* verlengt de tijd van de move-operatie zodanig, dat deze kan worden voltooid.

Excepties Het exceptie-support-mechanisme biedt de FU de mogelijkheid 3 types excepties te signaleren (zie sectie 3.3). Een herstelbare exceptie betekent dat de huidige transport-opdracht niet voltooid is, doch dat deze na afhandeling van de exceptie kan worden afgemaakt. Het *trap* en *return* mechanisme hangt af van de implementatie van de exceptie-FU. In het prototype wordt gebruik gemaakt van de *identifiers* op de move-identifier-bus, om binnen één cyclus het adres van de juiste exceptie-routine te kunnen vormen. Dit maakt zeer snelle emulatie van niet geïmplementeerde functionaliteit mogelijk. Ten behoeve van speculatieve executie is het mogelijk om niet-preciese excepties op een later tijdstip af te handelen. Dit is mogelijk door het toevoegen van een exceptie-bit aan de data. Het testen van de exceptie conditie vindt nu programmatisch plaats.

4 Vergelijking van MOVE met architectuur trends

De vorige twee secties beschrijven de recente architectuur ontwikkelingen en ons alternatief, de MOVE-architectuur. In deze sectie vergelijken we deze recente ontwikkelingen met de MOVE-architectuur vanuit verschillende gezichtspunten. De belangrijkste reden voor deze vergelijking is de rechtvaardiging van de characteristieken van de MOVE-architectuur zoals gepresenteerd in de vorige sectie. Vergelijkingen vinden plaats op de volgende gebieden: 1) utilizatie van de hardware; 2) geschiktheid voor aanpassing van de MOVE naar de eisen van specifieke toepassingen; 3) *pipelining* van funktie-eenheden; 4) implementatie konsekwenties van excepties; en tenslotte, 5) prestatie aspekten zoals snelheid en code-omvang.

4.1 Utilizatie van de processor hardware

Bij het waarderen van de MOVE-architectuur met betrekking tot hardware gebruik, moeten we kijken naar de verschillende componenten in een MOVE-processor. Zoals aangegeven in figuur 3a bestaat een MOVE-processor uit een transportnetwerk en funktie-units. Registers kunnen ook beschreven worden als funktie-units, maar we beschouwen deze apart.

Transportnetwerk In sectie 3.1 is uitgelegd dat een MOVE-processor de interne transportcapaciteit efficiënter gebruikt dan een met de MOVE vergelijkbare VLIW. Vergeleken met de VLIW, betekent dit dat we met dezelfde hoeveelheid interconnectiemetaal meer funktie-units kunnen aan spreken en bezig houden. Integratie op een enkele chip wordt hierdoor vereenvoudigd, omdat er minder interconnectie *overhead* is.

Funktie units Functie-units (FUs genaamd) zijn in de MOVE-architectuur bijna geheel gescheiden van het data-transportnetwerk. De FUs hoeven slechts te voldoen aan een FU-transport *interface* beschrijving. Dit betekent bijvoorbeeld dat we de *pipelining* van functie-units kunnen laten afhangen van de funktie en het gebruik. De enige restrictie die opgelegd wordt door het netwerk is de tijd tussen het triggeren van operaties; deze tijd is namenlijk een veelvoud van de data-transporttijd[5].

Register use Het aantal registers benodigd in een MOVE-architectuur kan sterk worden verminderd vergeleken met andere architecturen. Hiervoor zijn drie redenen aan te geven:

[5]Dit maakt zelfs *wavepipelining* mogelijk. Wavepipelines vereisen wel dat data op tijd wordt gelezen uit de pipeline. Ondersteuning van excepties betekent dan ook dat de pipeline moet uitlopen in een aantal zg. *uitloop* registers.

1. Minder tijdelijke waarden hoeven bewaard te worden. Veel van deze tijdelijke waarden gaan direkt van FU naar FU zonder in de *general-purpose-register* geschreven te worden.

2. Effectief blijkt, dat *Pipeline*secties en FU-operand en FU-resultaat registers worden gebruikt ter vervanging van *general-purpose-registers*. In de MOVE-architectuur wordt een gewone RISC-instruktie gesplitst in zijn drie fundamentele transportcomponenten en deze componenten worden apart van elkaar *gescheduled*. Een resultaat dat gegenereerd wordt door een FU, maar niet direkt gebruikt kan worden (door de zelfde of een andere FU), kan tijdelijk blijven staan in de genererende FU, behalve uiteraard, indien dit resultaten blokkeert die eerder nodig zijn. In dit laatste geval is wel een register nodig om dit resultaat even te bewaren. Op dezelfde manier kunnen we een FU-source-register gebruiken om een waarde tijdelijk in op te slaan voordat de operatie op deze source-operand wordt uitgevoerd.

3. Blokkerende excepties. Wij hebben gekozen voor het automatische blokkeren van de *pipeline-stages* binnen in een FU wanneer de resultaten nog niet zijn weggehaald. Deze keus heeft tot gevolg dat geen compilatie-strategie nodig is waarin registers voor lange tijd gereserveerd worden.

4.2 Prestatie

Voor de prestatievergelijking kijken we naar: cyclus-tijdbeperkingen, *scheduling* vrijheid, code grootte en de geschiktheid voor het verwerken van scalaire en vector toepassingen.

Cyclustijd De meeste processoren gebaseerd op de RISC-ontwerpfilosofie gebruiken verschillende *pipeline-stages* voor het ophalen van instrukties (IF), uitvoeren van de operatie (EX of ALU) en geheugen toegang (MEM). De cyclustijd van deze processoren wordt nu beperkt door de tijd benodigd voor *cache* toegang (IF en/of MEM) of voor de operatie (ALU). Zowel *cache* toegang als operatie kunnen worden gesplitst in meerdere *pipeline-stages* (*superpipelining*). De *cache* toegang kan opgesplitst worden in een decodeer-, een opzoek- en een *tag*-vergelijk-*stage*. Het is ook mogelijk om *interleaving* op de *cache* toe te passen indien de opzoek-*stage* het kritieke pad in de cyclustijd vormt (de opzoek-*stage* kan kritiek worden indien te veel *cache*-lijnen gelezen dienen te worden).

De operatie-*stage* bestaat uit het lezen van de *source-operand-latches*, het doorlopen van de kombinatorische logica, het doorlopen van de *bypass-multiplexers* en vervolgens schrijven van een *bypass-register* en eventueel van een source-operand-latch. In principe kunnen we ook een *pipeline-latch* plaatsen direkt na de logica en voor de *multiplexers*. Zoals eerder beschreven ([15]), is de minimale hoeveelheid logica tussen *staging*-registers afhankelijk van hoeveelheid data- en klok-*skew*. Het verband tussen de schrijftijd via de *multiplexer* en de schrijfbelasting is lineair (zie [16]). Deze belasting is evenredig met het aantal ingangen van de *multiplexer*. De schrijftijd wordt dus $\theta(n)$ waarin n het aantal *multiplexer* ingangen is[6]. Het schrijven van n mogelijke *sources* naar 1 bestemming zou wel eens het kritische pad kunnen worden in zwaar ge*pipeline*de processoren. Dit heeft als konsekwentie dat het lezen uit een *register-file* (hetgeen ook een n naar 1 transport is) of het voeren door een komplex *bypass* circuit weleens de cyclustijd kan gaan bepalen. Behalve ontwerpen voor een minimum aan *data*- en *klok-skew*, dienen zowel de grootte van de register-*file* als de grootte van de *bypass* te worden beperkt. Zoals reeds aangegeven, is de MOVE-architectuur superieur wat betreft het beperken van het aantal benodigde registers. Het *bypass* circuit is vervangen door het transport netwerk en het aantal verbindingen binnen het netwerk is veel minder in vergelijking met een VLIW architectuur. Het is onze overtuiging dat wanneer transport het kritischepad wordt in de cyclustijd, de MOVE-architectuur geen reële concurrentie heeft.

[6]Het gebruiken van grotere *drivers* heeft geen zin omdat kapaciteits-belasting evenredig toeneemt met de *driving*-kapaciteit . Het gebruik van een *driver* boom, beperkt de tijd tot $\theta(log(n))$. Een boom is echter veel moeilijker te bedraden in VLSI.

Scheduling-vrijheid De prestatie van een VLIW is op cruciale wijze afhankelijk van de kwaliteit van *scheduling* van de code. In de MOVE-architectuur is een typische VLIW-RISC-operatie gesplitst in één of meer data-transport operaties (*moves* genoemd). Een 3 operand RISC-instruktie, bijvoorbeeld, komt overeen met 3 *moves*; één voor het transport van de eerste operand (*operand-move*), één voor het transport van de tweede (en laatste) operand (*trigger-move*) en één om het resultaat van de operatie te transporteren naar het uiteindelijke doel (*result-move*). De instruktie die de laatste operand transporteert, veroorzaakt ook het opstarten van de operatie (*trigger*). In combinatie met hybride-*pipelines* (zie 3.2) in FUs, deze opsplitsing in transport komponenten veroorzaakt nieuwe vrijheidsgraden voor *scheduling* die nog niet eerder vertoont zijn. Zowel het transport van operanden als van resultaten zijn losgekoppeld van het transport van *trigger*-operanden. Het splitsen van een traditionele instruktie in zijn transportkomponenten heeft verschillende voordelen:

1. Het elimineren van gemeenschappelijke subexpressies (CSE) kan worden toegepast op het transport van operanden. Indien een operand niet verandert tussen twee operaties die gebruik maken van de zelfde FU, hoeft deze operand slechts één keer naar de FU getransporteerd te worden.
2. Het verwijderen van onnodig transport naar *general-purpose-registers* doordat een resultaat niet direkt geconsumeerd hoeft te worden. Wanneer, bijvoorbeeld, de tweede operand voor een operatie nog niet beschikbaar is moet de eerste operand bewaard worden in een *general-purpose-register*. Mits de FU die deze operand produceert niet onmiddelijk nodig is, kan binnen de MOVE-architectuur dit bewaren ook gebeuren in deze FU zelf door de *operand-move* uit te stellen.
3. Het verkrijgen van betere initiatie-intervallen in *software-pipelined-loops*. Zoals opgemerkt in [17] is het vaak wenselijk om vertragings-*stages* toe te voegen in *pipeline*-reserveringstabellen om optimale initiatie-intervallen te kunnen verkrijgen. Voor de MOVE betekent dit, dat het transport vanuit een resultaat-register vertraagd dient te worden. Ook hier geldt dat dit slechts kan voor die FUs die niet op volle snelheid gebruikt worden.

Code grootte De verschuiving van CISC naar RISC veroorzaakte een vergroting van de object-code van ongeveer 50%. Deze vergroting is acceptabel gezien 1) de capaciteitsvergroting van RAMs en 2) het gebruik van grote instruktie caches ter compensatie van het gestegen instruktieverkeer. Het gebruik van VLIW architecturen kan echter gemakkelijk leiden tot een code explosie vanwege 1) het (gemiddeld) grote aantal ongebruikte operatie-*slots* en 2) de code duplicatie vereist voor geadvanceerde compiler technieken. In principe zijn MOVE-instrukties zelfs minder compact dan de vergelijkbare RISC-instrukties. Immers een 3-operand RISC-instruktie vertaald in drie moves. In ons MOVE-prototype, een MOVE-instruktie kost 16 bits, hetgeen neerkomt op 48 bits voor drie moves of 50% extra instrukties. Gelukkig zijn er verschillende redenen waarom deze 50% een bovengrens is, die gemiddeld genomen veel lager uitvalt:

1. Veel *moves* transporteren data van FU naar FU. In het ideale geval veroorzaakt dit 1.5 *move* per 3 operand RISC-instruktie; een code reductie van 25%.
2. Er zijn instrukties met minder dan drie operanden (branch/jump/monadische).
3. Het elimineren van gemeenschappelijke subexpressies (CSE) verwijdert ook onnodige operand *moves*.

Metingen met behulp van onze prototype-compiler, welke slechts *scheduling* op *basic-block*-niveau uitvoerde zonder CSE, laten zien dat voor een aantal grote *benchmarks* het gemiddeld aantal moves per RISC-instruktie ongeveer 2.2 is. De volgende versie van de compiler zal hoogst waarschijnlijk de RISC-instructiedichtheid evenaren. Wij verwachten dat met betrekking tot code dichtheid, de MOVE-architectuur vergelijkbaar of zelfs beter is dan een RISC.

Vergeleken met een VLIW- de MOVE-architectuur is superieur met betrekking tot code dichtheid. Allereerst is het aantal ongebruikte instructie *slots* gereduceerd vanwege het *shared*-transport netwerk (zie sectie 3) en ten tweede, is het mogelijk om volledige lege instructies te vermijden met behulp van *pipeline-blocking* (hetgeen essentieel is voor scalaire code).

Scalair en vector toepasbaarheid De MOVE-architectuur is ontworpen om uitbreidbaarheid in aantal en type FUs mogelijk te maken. In combinatie met de bijna onbeperkte mogelijkheden voor het implementeren van FUs en het efficiente gebruik van het data-transportnetwerk is de MOVE-architectuur uitermate geschikt voor het gebruik in vector en andere speciale toepassingen. Eén van de problemen van veel vector machines is de onbalans tussen de vector en de scalaire prestatie. Amdahl's wet beperkt daarom de bruikbaarheid van deze machines voor algemene vector toepassingen. VLIW architecturen gecombineerd met slimme compilers zijn in staat om de prestatie van zowel vector als scalaire code te verbeteren. Alhoewel de prestatie verbetering voor scalaire code beperkt is tot een factor van 2 à 4.

De MOVE-architectuur verbeterd een standaard VLIW in 2 aspecten, 1) zoals reeds gezegd, de code uitbreiding is minder dramatisch en 2) extra FUs kunnen makkelijk toegevoegd worden zonder dat veranderingen in het transportnetwerk vereist zijn en zonder dat het instructieformaat verandert dient te worden. In standaard VLIWs worden minder gebruikte functies gecombineerd in een enkele FU (bijv. integer, logic, shift en branch). In MOVE kunnen al deze functies worden gescheiden zonder veel kosten (alleen extra operand latches). Deze scheiding heeft tot gevolg dat deze functies ook parallel gebruikt kunnen worden en daardoor tot een prestatie verbetering leiden voor de scalaire performance.

Vergeleken met een RISC-processor lijkt het net alsof de MOVE-architectuur een inherent nadeel voor scalaire code heeft. Zoals getoond in figuur 5, ontstaat tussen twee afhankelijke operaties een extra cyclus (*lost-cycle*). In beide gevallen is de hoeveelheid werk per instructie vergelijkbaar: propagatie door kombinatorische logica en het *latchen* van het resultaat (zie sectie 4.2). Het verschil ligt hem in het feit dat de MOVE-architectuur twee maal *latched* inplaats van één maal (meteen na de logica, en meteen na het transport). De FU-resultaat-*latch* kan echter wel gecombineerd worden met combinatorische logica (door gebruik te maken van Earl of Polarity-hold *latches*).

Het is ook mogelijk om het aantal cycli te verminderen op de zelfde manier als in een RISC door het verwijderen van de *trigger-latch* (zie figuur 6). Deze laatste oplossing voor de *lost-cycle* is echter tegenstrijdig met het streven de cyclustijd te minimaliseren en nuttige *moves* the plaatsen in de *lost-cycli*. Het is in dit geval dan ook te preferen om minder transport capaciteit te implementeren en zodoende een betere vulgraad voor deze lost-cycli te verkrijgen. Al met al, verwachten wij dat een MOVE-processor met twee *moves* per instructie reeds beter presteert dan een standaard RISC.

4.3 Toepasbaarheid voor het ontwerpen van applikatie-specifieke processoren

Uitgaand van *state-of-the-art* silicon-compilers, met faciliteiten voor het gebruik van geparametrizeerde VLSI cellen, wordt het snel kunnen wijzigen van een processor een reële mogelijkheid. Een processor is geschikt voor het aanpassen aan specifieke toepassingen indien de architectuur zowel flexibel als wel uitbreidbaar is en het ontwerpen eenvoudig is.

Flexibiliteit Het *retargeting*'s proces vereist dat de functionaliteit flexibel te wijzigen is. VLIWs hebben hier een duidelijk nadeel; het toevoegen van FUs behelst: 1) het veranderen van het instructieformaat en 2) het toevoegen van extra bussen (hetgeen de volledige *register-filelayout*verandert). De MOVE-architectuur heeft dit nadeel niet. De MOVE-architectuur legt ook geen beperkingen op aan het aantal operanden en resultaten van specifieke FUs.

Uitbreidbaarheid in prestatie Naast de mogelijkheid tot het op eenvoudige wijze veranderen en aanpassen van de functionaliteit is het van uitzonderlijk belang dat de prestatie aan te passen is aan de eisen van de toepassing. De MOVE-architectuur heeft hier een extra vrijheidsgraad; behalve het toevoegen van FUs, is het mogelijk om onafhankelijk de transportcapaciteit te vergroten. Het transportnetwerk kan bijvoorbeeld geimplementeerd worden met behulp van een aantal parallelle bussen die gegenereerd worden door parametrizeerbare VLSI bibliotheekcellen. Het toevoegen van een bus is eenvoudig, alhoewel deze toevoeging

het instructieformaat verandert. Deze verandering is echter voorspelbaar en daardoor ook eenvoudig in de compiler door te voeren (bijvoorbeeld het aantal *move* bussen als compiler parameter). Het veranderen van de *pipeline*-graad is een andere manier om de prestatie aan te passen.

Ontwerptijd Bij het ontwerpen van ASPs spelen twee kwesties een belangrijke rol: de prestatie-kosten verhouding en de ontwikkel-tijd (*time to market*); Ontwerptijd is van cruciaal belang voor beide kwesties. Een lange ontwerptijd veroorzaakt hoge ontwerpkosten en een lagere performance omdat het ontwerp op verouderde technologie wordt gebaseerd. Het aantal ontwerp beperkingen dat inherent door de MOVE-architectuur wordt opgelegd is echter bijzonder laag; het veranderen van FUs, het toevoegen van FUs en het veranderen van het transportnetwerk kan allemaal onafhankelijk van elkaar gedaan worden. In onze overtuiging is de MOVE-architectuur ideaal voor het snel genereren van ASPs, gebruik makend van bestaande VLSI ontwerpomgevingen.

5 Conclusies

1. Een architectuur van een Centrale Processor Unit (CPU) waarbij data-transport van operanden en alle, of de meest belangrijke, operaties op deze operanden ontkoppeld zijn. De operaties op operanden worden verzorgd door zg. functie-units (FUs); het data-transport vindt plaats over een data-transportnetwerk. Programmering van deze architectuur vindt plaats d.m.v. het specificeren van data-transportoperaties. Per instructie worden 1 of meerdere van deze transportoperaties gespecificeerd. Iedere transport operatie specificeert één source en één of meerdere bestemmingen.
2. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij voor het data-transport een hierarchische busstructuur gebruikt wordt.
3. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij de data-transport operaties conditioneel worden uitgevoerd.
4. Een architectuur zoals gespecificeerd onder conclusie 3, waarbij de condities gespecificeerd worden door 1 of meerdere guards.
5. Een architectuur zoals gespecificeerd onder conclusie 4, waarbij de specificatie van de guards onderdeel vormt van de specificatie voor de data-transport operaties.
6. Een architectuur zoals gespecificeerd onder conclusie 5, waarbij de specificatie van het gebruik van de guards uit boolean-expressies bestaat, met in deze expressie de specificatie van 1 of meerdere guards.
7. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij het data-transport geïmplementeerd is d.m.v. single-cycle data-transport instructies.
8. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij het data-transport gepipelined geïmplementeerd is. Het data-transport is hierbij gescheiden in een lees cyclus en een schrijf cyclus.
9. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij het data-transport geïmplementeerd wordt door middel van een combinatie van 1, 2 of meerder cycli data-transport operaties. Een data-transport operatie van 3 of meerdere cycli bestaat uit een lees cyclus 1 of meerdere transport cycli, en 1 schrijf cyclus. Data-transportoperaties geïmplementeerd d.m.v. meerdere cycli worden gebruikt om clusters van functie-units met elkaar te laten communiceren.
10. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij het transport-netwerk en/of één of meerdere FUs met *selftimed* logica zijn geïmplementeerd.
11. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij voor één of meer functie-units een zg. hybrid pipeline schema wordt gebruikt.
12. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij één of meer functie-units geïmplementeerd zijn m.b.v. wave pipelining.
13. Een architectuur zoals gespecificeerd onder conclusie 11 of conclusie 12, waarbij ten behoeve van excepties één of meerdere functie-units zijn uitgerust met uitloop registers. Deze functie-units zijn tijdens een exceptie direct bruikbaar voor de exceptie code. Na terugkeer vanuit een exceptie wordt tijdens het lezen van resultaten uit deze functie-units automatisch eerst de uitloop registers gelezen.
14. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij preciese-excepties worden ondersteund d.m.v. het zg. inexact exceptie mechanisme.
15. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij tijdens het lezen vanuit een functie-unit van een niet-precies exceptie resultaat niet direct een exceptie gegenereerd wordt, doch waarbij de exceptie als extra conditie met de data over het transport netwerk mee propageert. Op een later, door de compiler bepaald tijdstip kan deze exceptie worden afgehandeld.

16. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij binnen 1 cyclus de juiste exceptie-vector wordt bepaald. Hierbij wordt het transport identificatie-nummer van die transport operatie die een exceptie veroorzaakt gebruikt voor deze bepaling.

17. Een architectuur zoals gespecificeerd onder conclusie 16, waarbij niet geïmplementeerde functionaliteit d.m.v. een emulatie mechanisme wordt geëmuleerd.

18. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij het transport netwerk automatisch blokkeert zolang 1 van de data-waarden die getranporteerd dienen te worden, nog niet door de functie-unit, die deze waarde berekent, geleverd kan worden, omdat de berekening van deze data-waarde nog niet voltooid is. Zodra deze berekening voltooid is wordt de blokkering opgeheven.

19. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij een functie-unit waarnaaar een data-waarde getransporteerd dient te worden, het data-tranport netwerk blokkeert, zolang hij deze data-waarde nog niet kan verwerken.

20. Een architectuur zoals gespecificeerd onder conclusie 1, waarbij de instructie-unit zich als een standaard functie-unit gedraagt, d.w.z. gelijk aan de meeste overige functie-units. De operanden van deze functie-unit, het instructie-register en de programma teller zijn via het data-transport netwerk toegankelijk. Immediate data-waarden kunnen direct uit het instructie-register worden gelezen. Het tranport netwerk wordt beheerd door een z.g. buscontroler; deze heeft toegang tot het instructie-register en de programma teller via het data-transport netwerk of via een separaat transport netwerk.

21. Een architectuur zoals gespecificeerd onder conclusie 1 en conclusie 20, waarbij meerder instructie-units aanwezig zijn en met het data-transport netwerk zijn verbonden. De buscontroler kan instructies uit meerdere instructie-registers selecteren om deze te gebruiken voor de besturing van het data-transport netwerk.

6 Figuren

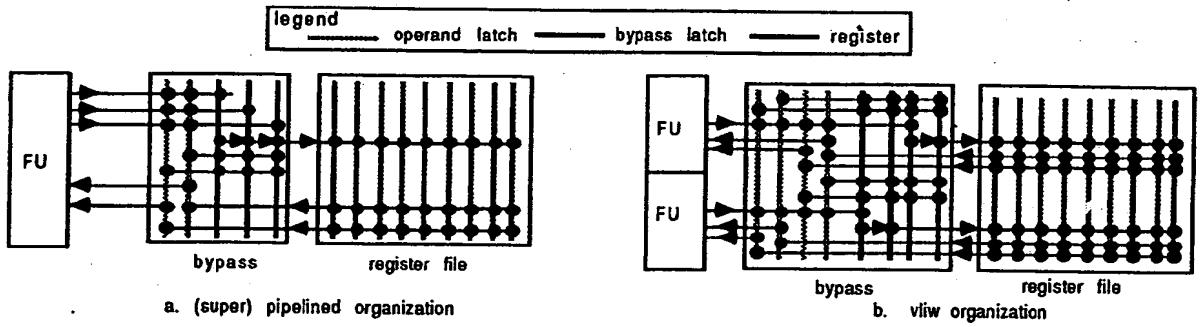

Figure 1: Register-FU-bypass communicatie voor (super) pipelined en VLIW organisaties.

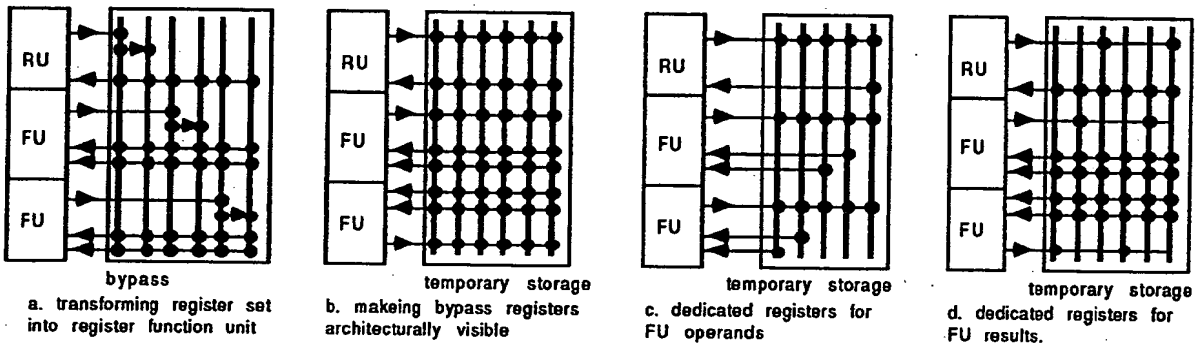

Figure 2: Maak de bypass zichtbaar op architectuur niveau.

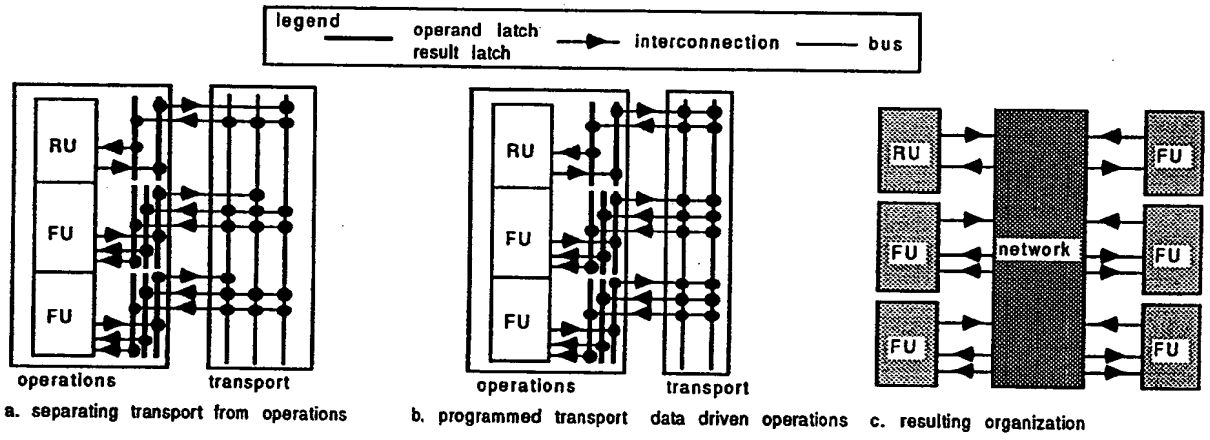

Figure 3: Separeer transport van operaties.

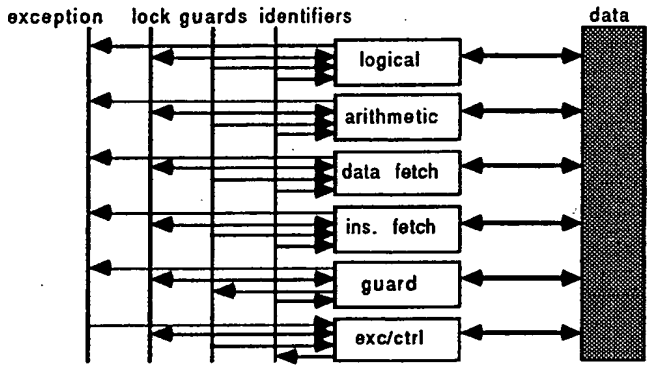

Figure 4: Support-mechanismen voor de MOVE-architectuur.

```
ADD     R1, R2, R3      R2 -> Radd-operand,   R3 -> Radd-trigger
SUB     R4, R5, R1      ........
                        Radd-result -> Rsub
                        .... etc.
```

Figure 5: Vergelijking van scalaire RISC- en MOVE-code

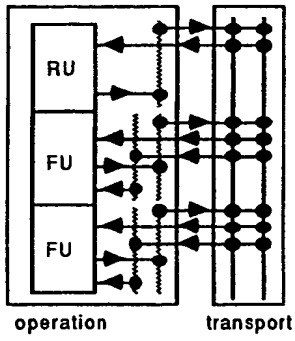

Figure 6: MOVE architectuur zonder de "verloren cyclus".